Computer Science • 20(2) 2019 https://doi.org/10.7494/csci.2019.20.2.3188

Marek Retinger

A CLIENT-BASED ENCRYPTION MODEL FOR SECURE DATA STORING IN PUBLICLY AVAILABLE STORAGE SYSTEMS

Abstract *This document presents a conceptual model of a system for protecting the data stored in publicly available data storage systems. The main idea was to apply encryption on both the client and server sides that would consequently have a significant impact on data security. The compatibility with existing systems allows us to deploy the solution fast and at a low cost. The tests conducted on a simplified implementation have confirmed the solution's validity, and they have shown some possible performance issues as compared to the classical system (which can be easily bypassed).*

Keywords data storage, data protection, encryption, security

Citation Computer Science 20(2) 2019: 179–194

1. Introduction

User privacy is one of the most popular topics these days. Most people do not want to share more data than necessary. The rising popularity of free storage services causes that more and more data is being stored in publicly available services. The huge amount of data that is being processed implies a great number of mistakes made by users (consciously or not) that may consequently lead to the disclosure of private and sensitive data. It is important to remember that data shared on the Internet once stays there forever.

Law [10] and users demand more sophisticated methods for secure data storage. The expectations are more and more restrictive, which can lead to many problems. Nowadays, one general solution does not exist.

The main goal of the research was to check the design and implementation possibilities of a system that could be used with existing solutions on the market that is cross-platform and ultimately increases ordinary users' data security. What is more, the data protection model should not allow us to process data on the server side or to obtain raw data even if the server's cryptographic key is stolen. The secondary goal was to define the system in such a way that it could work efficiently even on hardware with average performance. This causes the solution to become more useful, and its group of recipients increases.

The system described in the document discusses the case when the user stores the data by himself and any additional processing on the server side is not required. In real life, it could be useful for solutions where:

- no interaction between users occurs,
- no data sharing exists,
- no user data processing is required,
- problems between stored data and law may occur.

The proposed system application area is much wider. It can also be combined with existing software as an "extension" and can manage only some part of the stored data. In the future, new projects can be designed as described in the article in order to take full advantages of the solution.

2. Related work

The concept of partial or full database encryption is nothing new. In the past, many documents have been presented; however, no paper exists that describes a system similar to the presented one that works on both the client and server sides simultaneously without dedicated software on the user's computer.

In the past, some database encryption models have been proposed [18]. They mainly involve the protection of system tables, data transmission, and several strictly specified parts of database system such as views, triggers, and stored procedures. The idea is developed further. Today, each modern database management system includes independent encryption on the database, table, or column level [2].

With the rapid growth of cloud service popularity, researchers decided to fill the gap by introducing the concept of client-side encryption [4,17] based on homomorphic encryption, modern cryptographical algorithms, and third "well-trusted parties". The problem is the different performance on the client side; each user has their own hardware configuration and uses preferred software, which could cause performance issues. For the reason, a slightly different idea was shown in [12]. The researchers decided to introduce "encryption as a service" built based on XaaS (*everything as a service*) and a private cloud. This features many advantages, but it does not exhaust the topic of data encryption in public networks.

A most similar system was presented in [7]. The researchers used JavaScript scripts for client-side encryption and storing a ciphertext in an external database. They also used an HTML 5 feature – Local Storage. No more similarities were found; the differences are much more significant. In [7], a simple key-management system was introduced. It uses a file-based key and requires storing the key's hash in an external database. What is more, the raw user's password is not processed. It is necessary to enter approx. 43 bytes (chosen ASCII characters) – as mentioned in text, this could be very difficult to remember by ordinary users. The data-processing model is also different; it does not use a local in-browser database but an "encrypted object" [7]. The server uses one or more tables; it does not encrypt the data for a second time using the server's key. The dissimilarities are also visible in the libraries, tools, and environment that are used.

In all of the documents mentioned above, the approaches to security and data management are different. This results in the fact that no detailed comparison can be accurate.

3. Model overview

The proposed system is based on a client-side model of data processing; it has many features of a classical client-server model. The solution is detailed shown in a component diagram (Fig. 1) and activity diagram (Fig. 2).

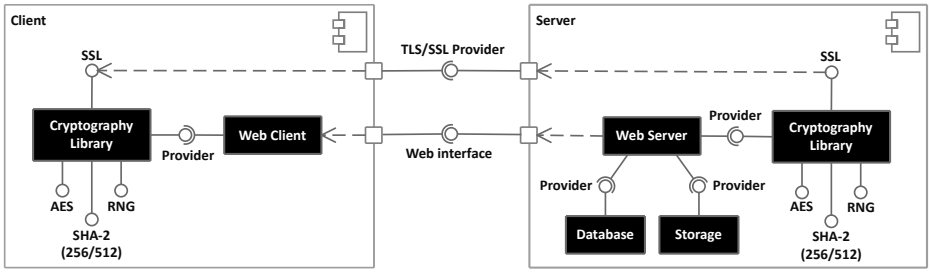

Figure 1. System's component diagram. Client's component does not contain local database module, as one is not required. *RNG* is random number generator. Connection between components can be done via any type of network

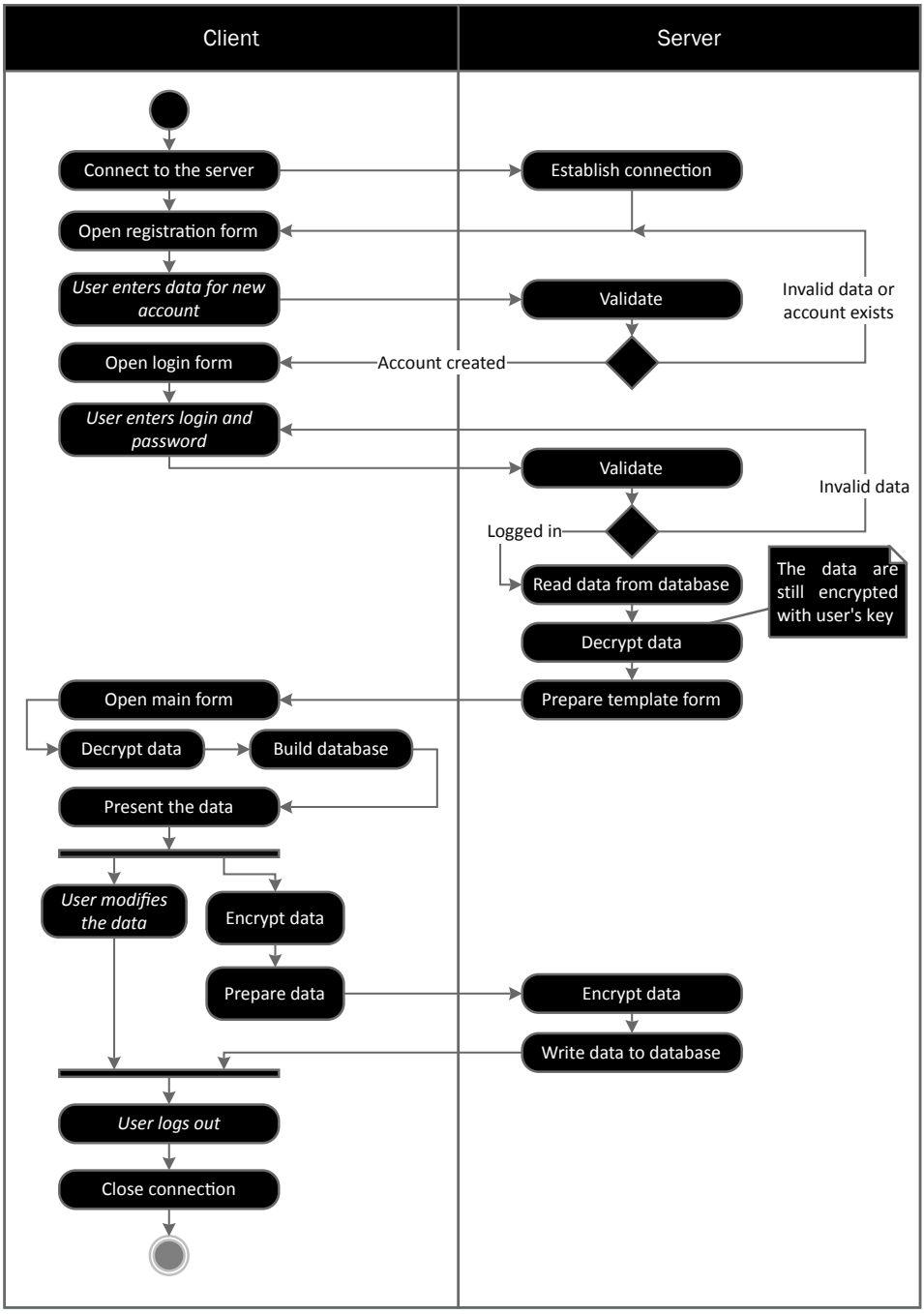

Figure 2. System's activity diagram. Entire single communication session was described (including account registration process)

The server (within the meaning of a service or a device) provides any kind of database engine and — if necessary — file storage. The database system is not strictly specified. The data can be stored in an advanced system such as a Microsoft SQL Server, simple SQLite library, or simply a structuralized file. All of the data is encrypted using a symmetric server key and random initial vector. The required pieces of data are a *resource identifier*, an *initial vector*, and the *encrypted data*. Importantly, the data is encrypted twice — on the client and server sides independently. This implies that compromising a server's data (including the server's secret key) should not affect the security of the stored data.

The client does not need to meet any special requirements. It is necessary to use an environment where basic cryptographical algorithms are available at a minimum, which allows us to store data securely. An additional advantage is local database support. The performance of the application depends on the chosen application type: stand-alone implementations in a high-level programming language will be most efficient at the expense of portability; a web page as a service will have the advantage in availability, but the data processing will be much slower.

The communication between the server and client is partially encrypted. Partial encryption means that the payload is protected but any additional information (for instance headers) is not. A good practice is to additionally use the SSL/TLS (*Secure Socket Layer/Transport Layer Security*) protocol [11], which encrypts all of the data transmitted.

In general, the client operates on the local copy of a database. When the edition is completed, the whole database is packed as a byte array, encrypted using the user's transformed key, and sent to a server. The server encrypts the data again using the server's key and stores the cryptogram in the database. Reading the data by the user is identical to above, but the process is done in reverse order. This is shown in Figure 3.

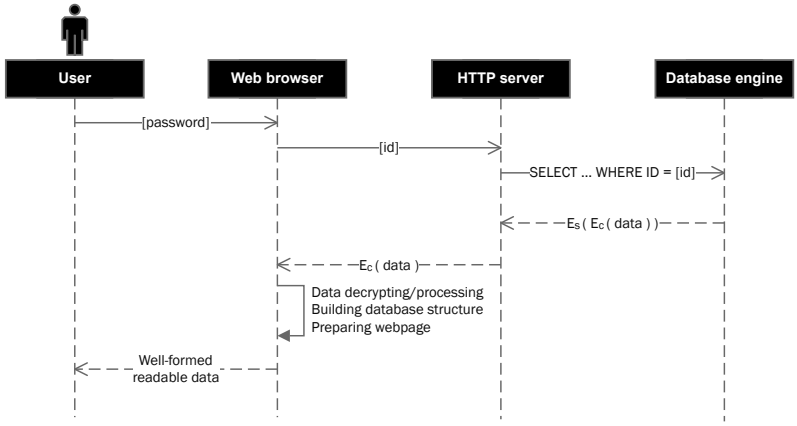

Figure 3. Reading data by user. Definitions of variables (in squared brackets) were presented in Section 3.1. E means symmetric cipher with client's key (c) or with server's key (s). SSL/TLS encryption layer intentionally omitted in scheme

3.1. System basis

The system works in several phases, which are described in detail below. In each phase, specific tasks must be performed, which requires that the proper order of an execution must be kept.

3.1.1. Registration phase

The starting state in which the user creates an account on the server. Firstly, the user provides basic information such as username and a strong (not trivial) password [3]. The client-side software builds a unique identifier *id*, which is sent to the server and validated there.

$$id = SHA_{dl}[login||KDF_{it_k,kl}(password)] \tag{1}$$

As shown in Formula (1), the password is processed by a key derivation function (KDF), where it_k is the number of iterations and kl is the output length. The concatenated *login* and KDF output are hashed by an SHA (*Secure Hash Algorithm*) function of a digest length equal to dl.

The client receives the server's response; if the *id* does not occur in the database[1], it obtains a raw local database structure. The structure is encrypted with a processed password (*ppassword*) and initial vector (*civ*), which is defined as shown in Formulas (2) and (3):

$$ppassword = KDF_{it_k,kl}(password) \tag{2}$$

$$civ = KDF_{it_{iv},kl}(password) \tag{3}$$

The encrypted structure is sent back to the server, encrypted using server's key, and stored in the server-side database.

3.1.2. Login phase

After the account-registration phase is finished, the user is able to obtain access to his account. He provides the username and password to the system. A client-side application transforms both values into an identifier as shown in Formula (1) and sends it to the server. What is important, the system is case-sensitive even for login; it cannot be resolved in any other way due to the use of the cryptographical hash function.

The server chooses the corresponding record in the database (if it exists) and decrypts it using the server's key. The server-side encryption is fully transparent for the client application. The data is further transmitted to the client and processed there.

[1]The *id* may exist in the database if the user uses the same login and password as somebody else or if a collision occurred.

The client decrypts the data using its own cryptographical key and uses plain text to create a local database structure that can be queried. Finally, a profile page can be prepared.

3.1.3. Work phase

The final phase is the "work phase". The user is able to use all of the features of the system. Each modification (for instance, adding a new item) is saved in the local database and has an impact on the visible content. Two possible approaches for saving data on the server are as follows:

1. data is saved manually by user — on demand;
2. data is saved automatically when specified period of time is exceeded.

In both cases, the local database is packed as a byte array, encrypted, and sent to the server, where the data is encrypted once more using the server's key. The result is stored in the database — the proper record is updated.

3.1.4. Lost password

Data recovery is impossible when a user forgets a password. One possible solution is a backup created by the system each time when changes are made. After account registration and initial profile configuration, the system must generate recovery keys (as Google allows) and save them in the local database. The user should also print or securely store them.

When the user logs out or a certain amount of time has elapsed, an n-copy[2] of the local database is sent to the server. If the main password is lost, the user has to login using one of the recovery keys. The problem that remains on the server side is the detection of inactive accounts. In the case where the user has recovered the data, the "old" database entry with the unreachable and unnecessary data will still remain on the server. The only way to omit the problem is to add an additional column in the server's database and to register the data access, which might not be accurate.

3.2. The implementation

The practical value of the proposed solution can be determined only when an implementation exists. It allows us to conduct tests and measure the performance in real-life cases. The implementation is based on the assumptions mentioned in Section 3. Intentionally, only well-tested algorithms were chosen; this guarantees the high performance and high security of the system.

For implementation purposes, a simplified version of the web service was built. The main goal was to create a website where a single user can create a private profile and store sensitive data for himself as text notes. The profile can be accessed by simply using a modern web browser without any add-ons. The client type was chosen

[2]The number of the copy depends on user's profile configuration.

intentionally — "in-browser" applications offer much worse performance than equivalent solutions in any high-level programming language [19]. The "web page" was built using technologies such as JavaScript, HTML 5, and CSS 3. The purpose of their use is described below.

- JavaScript — the application core was written with the use of several external libraries and built-in tools like WebCrypto [1] (provides cryptographical tools) and TaffyDB (provides database support). Encryption, decryption, the execution of queries, and the preparation of a user interface are done in JavaScript. The first version of the client is based on the CryptoJS library; however, it was replaced by a native web browser cryptographical tool due to its low performance.
- HTML 5 — the fifth version of HTML introduces local space management called Web Storage [9]. The most important and useful part of it (in the case of implementation) is Session Storage, which provides a protected memory area. The protected area is accessible locally and only for the tab that saved the data. In the client, it was used for storing a cryptographical key. HTML 5 was also used for front-end purposes.
- CSS 3 — used only for front-end purposes.

On the server-side, a Microsoft SQL Server was deployed as a database engine, and Python 3.5.2.3 was deployed as a run-time environment. The database stores one table containing three *varbinary*-type columns as mentioned in Section 3. The main implementation was written in the Python language using the Flask web framework. Due to some problems with the database server connector, the newest version of the ODBC (*Open DataBase Connectivity*) driver was used. The connection string was changed accordingly. For the encryption purposes, the PyCrypto library was chosen.

The cryptographical tools that were used are the AES (*Advanced Encryption Standard*) cipher working in CBC (*Cipher Block Chaining*) mode with a key length of 128 bits, the SHA hash function with an output length of 512 bits, and the PBKDF2 (*Password-Based Key Derivation Function 2*) [8] key derivation function. The numbers of iterations were 100 for the user key and 150 for the user initial vector. The output length for PBKDF2 was 128 bits.

4. Security and performance analysis

The system's performance was tested on three different platforms. Due to the fact that the results were strongly dependent on processor performance, the tests were conducted on different hardware as shown in Table 1. The client-side software was the same on each platform and is not mentioned in the table – Google Chrome 70 was chosen.

The "classes" were defined for testing purposes only. Modern high-end platforms were omitted; these are relatively uncommon [16], and the performance results would obviously be difficult to obtain by ordinary users.

Table 1
Specification of testing platforms

	Class 1	Class 2	Class 3
Processor	Intel Atom D525	Intel Core 2 Quad Q8300	Intel Xeon E3-1240v3
Memory	DDR3 / 4 GB	DDR2 / 6 GB	DDR3 / 8 GB
Storage device	WD 250 GB 5400 RPM	Plextor M5 Pro 128 GB	Samsung PM841 128 GB
Operating system	Microsoft Windows 7 x64	Microsoft Windows 7 x64	Microsoft Windows 10 x64

4.1. Performance analysis

The performance tests were divided into two main categories. The first one aggregates the tests in which the cryptography overhead was measured. The second category contains a performance comparison between the local "on-demand" database and the relational databases when specified queries are being executed. All of the tests were performed using the same database structure, which is shown in Figure 4. The amount of data stored in the database was 1 record in the profile table, 100 records in the album table, and 1 MB, 5 MB, or 10 MB in the photo table. The way the data is stored is described in Section 3. The local database structure is rebuilt after decrypting the retrieved ciphertext.

The main difference between the dataset categories is the photo's table size. The first group has a photo table truncated to reach the expected size of the database (1 MB, 5 MB, 10 MB). The second group uses only a 10-MB database.

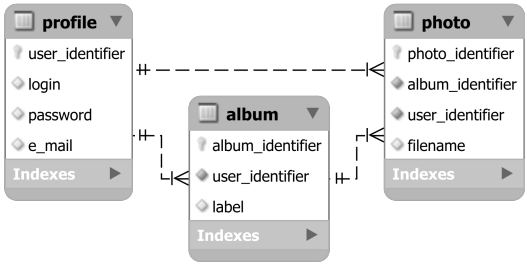

Figure 4. Sample database structure

The system used for the tests is described in Section 3.2 ("a"). There were two independent implementations written in the C# language, which used the SQLite database ("b") and Microsoft SQL Server 2012 ("c").

4.1.1. Cryptography overhead test

It is well known that the encryption and decryption processes are time-consuming. In the cryptography overhead tests, the system is compared to the implementations that do not use any cryptography techniques. During the tests, the data was retrieved from the main database and decrypted, and a local database was built. In all cases, only one simple query was executed. The results are listed in Table 2.

Table 2
Time required to obtain result

	Time [ms]								
	Class 1			Class 2			Class 3		
	a	b	c	a	b	c	a	b	c
Sample (1 MB)	291	230	280	210	129	205	126	54	125
Sample (5 MB)	1219	242	470	474	139	220	180	62	130
Sample (10 MB)	2634	247	531	1025	157	239	382	64	158

As expected, the execution of cryptographical algorithms is costly. The time difference between the encrypted and raw data increases in step with the size of the sample. In the cases of 1 MB and 5 MB, the required time is acceptable. Processing 10 MB takes a little more time; some workarounds could be applied as described in further sections. On the other hand, 10,240 kB is a lot of data for simple data types such as keys, notes, and calendars, which implies that this size should not be reached during a normal use. Thus, the performance drops should not be noticeable for a long period of time.

4.1.2. Performance test

The performance test scenario refers to a user who has a collection of photos. The information about the resources is stored in the database. In a probable scenario, the user will attempt to retrieve the gathered information about his resources. The performance was measured in three cases:

1. execution of a query that returns a list of all of the user's albums;
2. execution of a query that returns all photos from the album;
3. execution of a query that returns a number of photos in each of the user's albums.

As shown in Table 3, the solution shows worse results in the real-life examples than in the cryptography overhead test. This is caused by the client-side database, which requires all of the user's data to be stored in the server's database. The database library uses the JSON (*JavaScript Object Notation*) data format, which impacted the processing speed and size of the stored data (additional metadata is required). A lower system efficiency is mainly visible when a small amount of data is being processed; otherwise, the differences will be less significant. The solution efficiency can be improved by using the guidelines described in Section 4.2.

Table 3
Time required to obtain result

	Time [ms]								
	Class 1			Class 2			Class 3		
	a	b	c	a	b	c	a	b	c
Case 1 (10 MB)	4582	468	683	1932	182	271	864	43	98
Case 2 (10 MB)	8443	1579	1119	2801	416	321	1115	190	105
Case 3 (10 MB)	9186	3706	1370	2955	995	715	1373	467	274

4.2. Performance issues

Processing large volumes of data is not very efficient. The issue can be easily resolved in numerous ways:

- parallel encrypted and unencrypted databases,
- use of high-performance environments on the client-side,
- off-line work and on-demand synchronization,
- decomposition of a single database.

The last method is the most important solution. Decomposition of a single "cryptogram-based" column into several columns or tables resolves the problem for a long period of time. The decomposition forces minor changes. The data transmission should be done in two steps:

1. client receives root column, decrypts it, and determines which part of the data (from which columns) will be required; the request is sent to the server;
2. client receives the required columns, decrypts it, and executes the query.

In general, the performance is highly dependent on the chosen number of columns and their sizes. What follows is an example scenario:

- the data is stored in one column; the size of the data is 10 MB;
- the data is stored in five columns (one root column and four data columns); the size of the data stored in the root column is 20 kB, and the size of the data stored in each of the data columns is 2.5 MB.

Table 4
Size of data to be processed depends on number of required columns.

	Size
Root column only	0,02 MB
1 data column	2,52 MB
2 data columns	5,02 MB
3 data columns	7,52 MB
4 data columns	10,02 MB

As shown in Table 4, it is very important to create a proper structure of a local database. In the case where only two columns are required, only half of the data is

transferred; as a consequence, this reduces the processing time. In the worst case, it is required to download more data than in the single-column model; however, this situation is unlikely. The most important factor is the proper analysis of a target system and grouping the data by the frequency of use. This allows us to reach a high performance at a reduced amount of transmitted data.

4.3. Security analysis

The system security is guaranteed by cryptographical algorithms, which are known to be secure. The data on both the client and server sides are encrypted independently. An attacker who comes into possession of the whole database has to first find the server's secret key, which should be stored in a protected memory area. Even if the adversary finds or steals the key, he has to find the second (user's) symmetric key and corresponding initial vector for each entry in the database. As 128-bit-length keys are used, a significant amount of key space must be searched. Another way is to generate all combinations of the input phrases, which consist of lowercase and uppercase letters, digits, and special characters. The combinatorial space is huge, and any attack would not be efficient[3].

As local database may be represented as a structuralized form (for instance, JSON), and the first block of the cryptogram could be constant in some block cipher modes of operation. This is an undesirable effect. For text data encoded as UTF-8 and encrypted by a cipher with a block length of 128 bits, the first 16 characters will be considered, and these will impact the rest of the cryptogram. Two solutions could be used:

1. a random number of characters is put into the first line of the structuralized data; this requires the pre-processing of the data before the local database will be built;
2. a random key and value are inserted as a first element in the structuralized data; this does not require pre-processing, but several characters will first be known due to the publicly known structure.

The above steps are performed before the encryption phase and could be important in some implementations.

The analysis involves data transmission security. Communication between client and server should be protected by any known security protocol. The SSL/TLS protocol is preferred as the most common solution. The example mentioned below considers a scenario where the protocol cannot be used nor is it supported. The session cookie could be captured using a technique known as *cookie hijacking* [15]. In the case of the system, the cookie is not sufficient to obtain any user data. The attacker needs an additional cryptographical key to read the data. In some cases, the attacker is able to update the data (in the server side database) and corrupt the user's main profile.

[3]The minimal password length is eight characters.

However, there are still n copies of the user's main profile that can be *restored* using the user's recovery key(s).

The system was also briefly analyzed for well-known attack techniques such as Clickjacking [14] and XSS (*Cross-Site Scripting*) [6] attacks. Both of these are related to implementation vulnerabilities, which does not prove that the system's basis is incorrect. The first group of attacks involves scenarios when a user clicks on the specified element on the web page but the behavior is different than expected. The simplified situation could be when a user opens a link that leads to an infected website that was sent by a trusted user. Clicking a button that should perform the intended action on the website in fact removes the user's account from the other website in which he was logged. The attack specification and methods of defense were clearly described in [13]. The second group — also called the JavaScript code injection attack – is a client-side attack performed using server-side weaknesses. The adversary locates vulnerable websites and infects them with a malicious script. When a user opens the website, the script is executed, and unpredictable actions are performed. The situation is very dangerous, because data such as cookies, passwords, or any other confidential information can be stolen without the owner's knowledge. A basic example could be a scenario in which an attacker finds out that a form of a guest book module on a website is unprotected and the entries are published immediately. He prepares and injects a malicious script, which starts mining cryptocurrency in the web browser of each visitor. When any user opens the website's guest book, he becomes an unconscious worker of the adversary for the time he spends visiting the web page. In this case, no information is stolen, but the client's hardware resources are used without the website owner's or user's consent. Detailed information about XSS can be found in [6].

The implementation is resistant to known attacks from the groups by the following:

- using a special JavaScript code that disallows opening a website in a frame (HTML iframe element),
- filtering all input data and taking the proper actions (e.g., replacing special characters to its entities or removing HTML and JavaScript tags),
- setting a flag for session cookies as *HttpOnly*, which prevents them from being read from JavaScript,
- using an HTML 5 feature called Session Storage, which cannot be read from any other web browser tabs/windows than the primary one.

Users can also increase the level of the client-side security themselves by using web browser add-ons that can block unwanted and unsafe elements of websites (mainly scripts). Their names have intentionally not been mentioned here.

The performed tests confirm the system's resistance to chosen attacks. A security aspect summary is placed below.

4.3.1. Confidentiality

Confidentiality is ensured by a double-encryption process using a symmetric cipher that is known to be secure. The operation is done independently on both the client and server sides. The client uses his own key transformed by a derivation function to the secured key. Based on the output, the initial vector (IV) is built. The IV value is dependent on number of iterations (which is customizable by the user). The server uses an internal key for all stored data. The server's IV is different for each entry in the server's database and it is generated on demand. It means that even copies of a user's profile are not encrypted by the same IV.

4.3.2. Integrity

Integrity is ensured indirectly; there are no signed checksums. The data is encrypted twice, which causes that any modification of the data requires the possession of both keys (or only one if the attacker has taken full control of the server). Storing encrypted data prevents data processing; it forces "blind" bit (byte) manipulation in order to make any changes. Blind, because symmetric ciphers use a global diffusion block(s), which destroys the correlation between the input and output data. This means that any changes (without knowing a valid key) will be seen by the client as an error during the decryption process.

4.3.3. Availability

Availability is ensured in two ways. The first way is an automatic backup system. The user decides how many copies should be created; this number should not be less than two. The solution is still insufficient if a server would be inaccessible. The second solution is to use a platform in a cloud. The nature of cloud systems is to provide a high level of availability by distributing the data.

5. Future work

The proposed system is not fully mature yet. The next step will be a further development and becoming a part of a secure data storage system [5]. The main goals are to identify and resolve most of the performance and security issues. This may increase the popularity of client-based encryption.

6. Conclusion

The security of data (in particular, that which is stored in publicly available storage systems) is very important. There is no system that guarantees full security, even if a service provider states otherwise. A good solution is to protect the data personally; an even better solution is when the system does it for us, providing built-in tools.

The presented model of data protection is one step further and can improve a user's data security. The system is fully transparent for ordinary users, which

allows us to deploy the solution regardless of one's computer skills. The underlying cryptography algorithms can be replaced in an easy way and can be adapted to current standards. The compatibility with existing systems allows us to deploy the solution fast and at a low cost.

The system performance is on an acceptable level; however, it could be improved as described. As the tests have shown, the solution is usable even with older and low-end devices. It could also be provided as a web application, which additionally increases the value of the proposed system and the number of targeted users.

Acknowledgements

This work was supported by the 04/45/DSPB/0197 PUT grant.

References

[1] Cairns K., Halpin H., Steel G.: Security Analysis of the W3C Web Cryptography API. In: *Security Standardisation Research*, pp. 112–140, Springer, 2016. https://doi.org/10.1007/978-3-319-49100-4_5.

[2] Cebollero M., Natarajan J., Coles M.: *Pro T-SQL Programmer's Guide*, Apress, 2015. https://doi.org/10.1007/978-1-4842-0145-9.

[3] Dell'Amico M., Michiardi P., Roudier Y.: Password Strength: An Empirical Analy-sis. In: *2010 Proceedings IEEE INFOCOM*, IEEE, 2010. https://doi.org/10.1109/infcom.2010.5461951.

[4] Gaur T., Sharma D.: A Secure and Efficient Client-Side Encryption Scheme in Cloud Computing, *International Journal of Wireless and Microwave Technologies*, vol. 6(1), pp. 23–33, 2016. https://doi.org/10.5815/ijwmt.2016.01.03.

[5] Grocholewska-Czuryło A., Retinger M.: Secure cloud services – extended cryptographic model of data storage, *Przegląd Elektrotechniczny*, vol. 1(3), pp. 164–169, 2018. https://doi.org/10.15199/48.2018.03.33.

[6] Gupta S., Gupta B.B.: Cross-Site Scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art. In: *International Journal of System Assurance Engineering and Management*, vol. 8(S1), pp. 512–530, 2015. https://doi.org/10.1007/s13198-015-0376-0.

[7] Kaczmarczyk V., Bradáč Z., Fiedler P., Arm J.: Client side data encryption/decryption for web application. In: *IFAC-PapersOnLine*, vol. 49(25), pp. 241–246, 2016. https://doi.org/10.1016/j.ifacol.2016.12.041.

[8] Kaliski B.: PKCS #5: Password-Based Cryptography Specification Version 2.0, RFC 2898, 2000. https://doi.org/10.17487/RFC2898.

[9] Laine M.: Client-Side Storage in Web Applications, Aalto University, 2012.

[10] Layton R.: How the GDPR Compares to Best Practices for Privacy, Accountability and Trust, *SSRN Electronic Journal*, 2017. http://dx.doi.org/10.2139/ssrn.2944358.

[11] Lee H.K., Malkin T., Nahum E.: Cryptographic strength of SSL/TLS servers. In: *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement – IMC '07*, ACM Press, 2007. http://dx.doi.org/10.1145/1298306.1298318.

[12] Rahmani H., Sundararajan E., Ali Z.M., Zin A.M.: Encryption as a Service (EaaS) as a Solution for Cryptography in Cloud, *Procedia Technology*, vol. 11, pp. 1202–1210, 2013. http://dx.doi.org/10.1016/j.protcy.2013.12.314.

[13] Rydstedt G., Bursztein E., Boneh D., Jackson C.: Busting frame busting: a study of clickjacking vulnerabilities on popular sites. In: *In IEEE Oakland Web 2.0 Security and Privacy Workshop*, p. 6. 2010.

[14] Sankara Narayanan A.: Clickjacking Vulnerability and Countermeasures, *International Journal of Applied Information Systems*, vol. 4(7), pp. 7–10, 2012. http://dx.doi.org/10.5120/ijais12-450793.

[15] Sivakorn S., Polakis I., Keromytis A.D.: The Cracked Cookie Jar: HTTP Cookie Hijacking and the Exposure of Private Information. In: *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016. http://dx.doi.org/10.1109/sp.2016.49.

[16] Software – Avast PC Trends Report (Q3 2017), 2017. https://press.avast.com/hubfs/media-materials/kits/PC-trends-report-Q3-2017/avast_q3_2017_pc_trends_report.pdf.

[17] Souza S.M.P.C., Puttini R.S.: Client-side Encryption for Privacy-sensitive Applications on the Cloud, *Procedia Computer Science*, vol. 97, pp. 126–130, 2016. http://dx.doi.org/10.1016/j.procs.2016.08.289.

[18] Stokłosa J., Bilski T., Pankowski T.: *Bezpieczeństwo danych w systemach informatycznych*, Wydawnictwo Naukowe PWN, 2001.

[19] Zakas N.C.: *High Performance JavaScript: Build Faster Web Application Interfaces*. YAHOO PR, 2010. https://www.ebook.de/de/product/9283796/nicholas_c_zakas_high_performance_javascript_build_faster_web_application_interfaces.html.

Affiliations

Marek Retinger
Poznan University of Technology, Institute of Control, Robotics and Information Engineering, ul. Piotrowo 3a, 60-965 Poznan, Poland, marek.retinger@put.poznan.pl,
ORCID ID: https://orcid.org/0000-0003-3592-0942

Received: 26.02.2019
Revised: 15.05.2019
Accepted: 16.05.2019